



FONDAZIONE ISTITUTO TECNICO SUPERIORE  
PER LE TECNOLOGIE DELL'INFORMAZIONE E DELLA COMUNICAZIONE

TECNICO SUPERIORE PER I METODI E LE TECNOLOGIE PER LO SVILUPPO DI SISTEMI  
SOFTWARE  
WEB & MOBILE APP DEVELOPMENT

D68444-2-2018-0

**“ANOBII COVER SUGGESTIONS ENGINE”**

Candidato:  
**Tugnetti Simone**

in collaborazione con:



MINISTERO DELL'ISTRUZIONE, DELL'UNIVERSITÀ E DELLA RICERCA



per una crescita intelligente,  
sostenibile ed inclusiva  
[www.regione.piemonte.it/europa2020](http://www.regione.piemonte.it/europa2020)  
INIZIATIVA CO-FINANZIATA CON FSE

# Indice

<b>1. Introduzione</b>	<b>3</b>
1.1. Presentazione Biografica	3
1.2. ITS - Information and Communication Technology	4
1.3. Azienda Ospitante - Ovolab S.r.l	4
<b>2. Presentazione del Progetto Scelto</b>	<b>6</b>
2.1. Anobii Cover Suggestion Engine	6
2.2. The Main Search	7
2.2.1. Requisiti	7
2.2.2. Funzionamento	11
2.2.3. Esempi Layout	12
2.2.4. Esempi Codice	13
2.3. Barcode Scanner	14
2.3.1. Requisiti	14
2.3.2. Funzionamento	16
2.3.3. Esempi Layout	17
2.3.4. Esempi Codice	18
2.4. Image Management	19
2.4.1. Requisiti	19
2.4.2. Funzionamento	21
2.4.3. Esempi Layout	22
2.4.4. Esempi Codice	23
2.5. Cloud Images	24
2.5.1. Requisiti	24
2.5.2. Funzionamento	25
2.5.3. Esempi Layout	26
2.5.4. Esempi Codice	27
2.6. Suggestions History	28
2.6.1. Requisiti	28
2.6.2. Funzionamento	30
2.6.3. Esempi Layout	31
2.6.4. Esempi Codice	32
2.7. Linguaggi e Tool utilizzati	33
<b>3. Conclusioni</b>	<b>34</b>
<b>4. Sitologia</b>	<b>35</b>
<b>5. Ringraziamenti</b>	<b>37</b>

# 1. Introduzione

## 1.1 *Presentazione Biografica*

Mi chiamo **Simone Tugnetti**, 23 anni e da sempre appassionato di informatica e della scoperta dei vari linguaggi di programmazione.

Fin dalla più tenera età, il mio interesse per tale argomento è rimasto sempre vivo ma l'approccio verso tale realtà è stato molto difficile ai primi passi.

Solo quando mi fu regalato un computer non funzionante, provando più e più volte di ripararlo ed infine riuscendoci, capii di non abbandonare tale passione ma, al contrario, di perseguiirla verso un percorso di studi in grado di formarmi su quello che sarei voluto diventare, cioè un perito informatico.

Iniziai così un percorso all'interno dell'Istituto Tecnico "**I.I.S J.C Maxwell**", dove mi offrirono un percorso di studi incentrato sull'informatica generale, cioè vari corsi di programmazione, ad esempio Java, C++ e Javascript, e corsi inerenti ai sistemi informativi, come reti di calcolatori e Tecnologie dei Sistemi e delle Telecomunicazioni, cioè software comunicanti in rete.

All'ottenimento del diploma, decisi di non fermare il mio percorso, ma di migliorarlo, accrescendone la cultura e approfondendone gli argomenti.

Entraì quindi all'Università degli studi di Torino, nell'articolazione di Scienze della Natura, ambito Informatica, dove ho imparato alcuni approfondimenti, quali il linguaggio macchina, cioè Assembler, e le modalità avanzate di programmazione ad oggetti in Java, cioè i Nodi e i Thread Asincroni.

Mi resi conto ben presto però che tale percorso di studi, nonostante i corsi informatici, comprendeva una mole di corsi puramente matematici, i quali erano diventati molto più complicati da sostenere.

Feci quindi una scelta molto difficile, decidere che cosa fosse meglio per me, se continuare un corso universitario in più anni del previsto il quale stava calando di interesse, oppure cercare un'alternativa che mi avrebbe permesso sia di specializzarmi, oltre ad accrescere la mia conoscenza in un ambito specifico all'interno del contesto informatico, e sia di poter trovare alla fine di tale corso un'occupazione per qualcosa di cui ho avuto da sempre la passione.

Ci furono molte ricerche, ma alla fine trovai una scuola in grado di soddisfare le mie esigenze.

## *1.2 ITS - Information and Communication Technology*

Iniziai così un nuovo percorso formativo all'interno dell'**Istituto Tecnico Superiore in Information and Communication Technology**, tale corso permette di specializzarsi in un ambito specifico per poi migliorarsi esponenzialmente tramite il lavoro svolto in azienda.

Vi è, quindi, la scelta effettiva dell'ambito in cui ci si vorrebbe qualificare in base alle proprie passioni ed esigenze tra Visual Design, IT - Security, Sviluppo Backend e Frontend.

Scegliere quale percorso di studi dedicare la mia formazione è stato complesso, dato che avevo un particolare interesse sia per il lavoro Backend, il quale permette lo sviluppo di codice lato Server per la buona riuscita di applicativi in grado di far comunicare l'utente con i servizi online, sia per il lavoro Frontend, il quale, invece, permette lo sviluppo di applicativi WEB e Mobile, come Android ed iOS, più vicini alla parte funzionale e visiva verso l'utente finale.

Alla fine, scelsi lo sviluppo **Frontend**, dato il mio forte interesse verso lo sviluppo Mobile, prima scoperto verso lo studio e poi migliorato da autodidatta.

Procedendo con il corso, nonostante qualche difficoltà iniziale, capii di aver fatto la scelta giusta, cominciai ad affinare la mia tecnica di sviluppo sulla programmazione e ad ampliare la mia conoscenza riguardo il Marketing digitale.

Verso la metà del secondo anno, iniziò la ricerca dell'azienda più giusta dove svolgere lo stage curricolare e, dopo numerose ricerche e curriculum inviati, trovai la più adatta alle mie esigenze.

## *1.3 Azienda ospitante - Ovolab S.r.l*



Si tratta di **Ovolab S.r.l.**, fondata nel 2002 da Alberto Ricci come compagnia incentrata nel software development con l'intento di creare grandi prodotti per gli utenti Macintosh<sup>1</sup>. Nel corso degli anni, diventò sempre più influente ed espansiva e tutt'oggi si occupa di programmazione di applicativi mobili per sistemi iOS ed Android.

---

<sup>1</sup> Computer prodotti dalla Apple Inc.

Ha inoltre lavorato con alcune delle agenzie più rinomate, come “La Repubblica.it”, “Galbani” e “Deejay”, oltre ad aver acquistato un social network inerente alla condivisione di interessi e passioni riguardo ai libri, chiamato “**Anobii**”.

Prendendo in considerazione queste informazioni, scelsi di eseguire il periodo di stage presso quest’azienda data la sua predisposizione verso la programmazione nell’ambito mobile e di come sarei potuto migliorare capendo nuove modalità di utilizzo su tale tecnologia.

Rimase però ancora un’incognita, scegliere l’ambiente di sviluppo, se iOS o Android. La risposta fu più semplice del previsto, decidetti **Android**, non solo perché nutro un particolare interesse verso tale sistema, nonostante iOS abbia anche il proprio fascino, ma anche per gli anni spesi precedentemente, prima come autodidatta e poi come assiduo studente, per imparare questo genere di ambiente.

Ero consapevole di dover imparare un nuovo linguaggio oltre a quello di cui ero già a conoscenza, cioè passare da Java a **Kotlin**, per lo sviluppo di questo tipo di progetto, ma non l’ho paragonato per nulla ad un ostacolo, ma anzi ad un nuovo inizio, una nuova opportunità per imparare qualcosa che non solo mi avrebbe aiutato ad aumentare le mie conoscenze informatiche ma anche nel capire le differenze di come poter progettare e successivamente creare una stessa applicazione utilizzando due linguaggi differenti ma simili tra loro.

Iniziai così questo nuovo percorso all’interno di una vera azienda e, dopo le principali introduzioni ed assieme ai membri aziendali, decidemmo subito quale progetto sarebbe stato il più esaustivo e duraturo, dato che si volle optare per un utilizzo anche all’interno dell’ambiente lavorativo e non solo fine a se stesso.

Venne così scelto un progetto che fosse stato in grado di comunicare con il social network da loro appena acquistato, cioè “Anobii”, in grado di offrire un servizio per i propri membri aziendali.

## 2. Presentazione del Progetto Scelto

### 2.1 *Anobii Cover Suggestion Engine*

Il progetto concordato in questione è un applicazione **Android** nativa, sviluppata in **Kotlin**, che permette il suggerimento di immagini come possibili cover per un libro specifico presente all'interno del social network “**Anobii**”.

Il funzionamento di tale applicazione viene gestito in modo tale che l'utente debba inserire il codice ISBN<sup>2</sup>, il titolo oppure il nome dell'autore di un libro al quale si voglia suggerire un'immagine, verrà quindi eseguita una ricerca per verificare se all'interno dei servizi per il reperimento dei dati sarà presente quello stesso libro ricercato e che possieda almeno un'immagine di copertina.

Nel caso in cui non si volesse inserire alcun dato sul libro, per mancanza di tempo o altro, sarà possibile scansionare il codice a barre sul retro del suddetto utilizzando la fotocamera del dispositivo per reperirne il codice ISBN da utilizzare per la ricerca.

Nel momento in cui tale ricerca sarà effettuata, in caso di successo, verranno visualizzate una o più cover da suggerire con il rispettivo servizio al quale sono collegate, a questo punto basterà selezionare quella desiderata per poi avere la possibilità di eseguire diverse operazioni con essa, come salvarla sul dispositivo fisico o in Cloud<sup>3</sup>, oltre a poter scegliere se suggerire un'immagine già salvata in entrambe le locazioni e, nel caso del Cloud, di poter prelevare od eliminare tali immagini, oppure di poter suggerire l'immagine visibile come cover per il libro richiesto.

Nel momento in cui avviene il suggerimento, verrà eseguito un salvataggio in Cloud contenente tutte le informazioni principali riguardanti quel determinato libro come il titolo, l'autore e l'anno di pubblicazione, oltre al numero di richiesta e, ovviamente, l'immagine suggerita, oppure, nel caso vi sia un errore durante il collegamento, vi è la possibilità di salvare tali dati in un Database locale.

La scelta di rendere questa applicazione “**condivisa**”, cioè con i dati disponibili in Cloud, reperibili e ben visibili, sta nel motivo in cui essa può essere utilizzata dai membri di Ovolab per essere consci di quali cover possano già essere state suggerite dai colleghi oppure condividere alcune immagini, così da renderle disponibili ad altri per averne un'ulteriore opinione, nel caso in cui una specifica immagine voglia essere suggerita ma si voglia anche sapere cosa ne pensa il resto del team, oppure, più semplicemente, per salvarle e reperirle in un secondo momento.

---

<sup>2</sup> Codice Identificativo Univoco per il singolo libro

<sup>3</sup> Server Firebase in remoto

## 2.2 *The Main Search*

### 2.2.1 *Requisiti*

In questo paragrafo verranno elencati gli argomenti principali di cui esserne a conoscenza per il corretto funzionamento e comprensione della schermata in questione.

- **Android Jetpack**

**Android Jetpack**, più comunemente definito **AndroidX**, è un insieme di componenti, strumenti e linee guida che riunisce la **Support Library**, cioè una collezione di librerie per fornire funzionalità più recenti su versioni precedenti di Android, con gli **Architecture Components**, cioè una raccolta di librerie che comprendono dalla gestione del ciclo di vita dei componenti dell’interfaccia utente fino alla gestione della persistenza dei dati, disponendoli in quattro categorie:

**Architettura, Fondamento, Interfaccia Utente e Comportamento.**

Un’applicazione creata utilizzando Android Jetpack non ha la necessità di utilizzare ogni singolo componente per il suo corretto funzionamento, dato che queste sono tutte librerie “separate”, reperibili all’interno di androidx.\*, dando quindi la possibilità di utilizzare solo quelle di cui vi è necessità.

Inoltre, grazie al supporto di Android KTX, cioè librerie Kotlin Extensions, viene applicata l’esecuzione di determinate operazioni utilizzando delle estensioni per creare un codice più semplificato e conciso rispetto ad un utilizzo normale, ad esempio utilizzando le funzioni Lambda per integrare l’evento di pressione di un pulsante.

- **Navigation**

Una **Navigation**, parte della categoria **Architettura** in **Android Jetpack**, è un componente utilizzato per rendere la navigazione all’interno dell’applicazione in modo più grafico e dettagliato.

Più precisamente, quando si deve passare da una schermata ad un’altra, di solito, si fa uso di **Intent** o **Fragment Transaction** in base alle necessità, questo però rende il cambiamento di posizione confuso e senza una base solida, dato che per essere sicuro di dove l’utente possa andare da e verso la schermata attuale, bisogna avere una conoscenza avanzata del progetto al quale si sta lavorando, se non esserne i fautori.

La Navigation arriva quindi in aiuto, dando un grafico dettagliato di come l’applicazione possa viaggiare da una schermata ad un’altra.

Questo componente, per eseguirne il corretto funzionamento, deve essere composto da quanto segue:

- **Grafico di navigazione**: una risorsa XML che comprende destinazioni, cioè le varie schermate, e percorsi, cioè l'azione da intraprendere per passare a tale schermata.
- **NavHost**: un contenitore vuoto che visualizza le destinazioni del grafico di navigazione.
- **NavController**: un oggetto che gestisce la navigazione dell'app all'interno di NavHost, qui possono essere decisi il percorso da utilizzare, la destinazione, le animazioni, ecc..., oppure utilizzare la modalità **Safe Args**, che permette di scegliere una direzione avendo una conferma ulteriore sul tipo di dato che si intende passare da una schermata ad un'altra.

## ● **ViewModel e LiveData**

Una classe **ViewModel**, parte della categoria **Architettura in Android Jetpack**, viene progettata ed implementata per permettere di archiviare e gestire i dati relativi all'interfaccia utente consapevolmente al ciclo di vita dell'applicazione.

Più precisamente, quando l'interfaccia utente di una determinata schermata subisce un cambiamento, come ad esempio la rotazione dello schermo o il passaggio in un'ulteriore schermata, i dati associati normalmente verrebbero eliminati oppure salvati all'interno di un metodo specifico chiamato **onSavedInstanceState()** per poi essere ripresi in un secondo momento, questo però risulta essere corretto solo per una piccola quantità di dati e non, ad esempio, per delle collezioni.

Con i ViewModels, invece, questi dati vengono gestiti in modo tale che quando si eseguono dei cambiamenti, le informazioni inerenti alla UI<sup>4</sup> verranno salvate direttamente all'interno della classe specifica per poi reperirla e visualizzarla appena il cambiamento risulta terminato, tutto questo legando la stessa classe al Lifecycle associato alla schermata.

I dati possono essere salvati all'interno di variabili della classe ViewModel aventi come tipo:

- **LiveData**: anch'esso parte della categoria **Architettura in Android Jetpack**, è una classe di detentori di dati che può essere osservata all'interno di un ciclo di vita. Ciò significa che, prendendo in considerazione un determinato Lifecycle, la suddetta classe può osservare lo stato dello stesso e attuare le modifiche solo se risulta essere attivo, nel caso in cui, invece, risulterà distrutto, si distruggerà anche l'osservatore non salvando di conseguenza i dati raccolti. Usata per reperire gli stessi dati.
- **MutableLiveData**: come la classe LiveData, con l'aggiunta che i metodi di settaggio dei valori sono pubblici e quindi direttamente modificabili.  
Di solito usata per modificare i dati al momento opportuno.

---

<sup>4</sup> User Interface

## ● Data Binding

Il **Data Binding**, parte della categoria **Architettura** in **Android Jetpack**, è una libreria utilizzata per associare i componenti dell’interfaccia utente presenti all’interno dei layout alle origini dati dell’app utilizzando un linguaggio dichiarativo anziché programmatico.

Questo metodo, infatti, viene utilizzato per visualizzare i dati all’interno di uno specifico **ViewModel**, dichiarando una variabile all’interno del layout associata ad esso, per poi richiamare la variabile contenuta nello stesso direttamente come campo di un determinato widget, ad esempio il testo di una `TextView`, aggiornandolo ogni qualvolta si modificherà la variabile, senza dover scrivere una singola riga di codice nel sorgente.

Esiste anche un metodo per poter creare funzioni apposite, da richiamare all’interno del layout, che eseguano una qualsivoglia operazione al fine di gestire metodi unicamente legati alla UI denominato **Binding Adapter**.

Un Adapter può essere utilizzato sia da un qualsiasi widget, creandone semplicemente una funzione al quale passarlo come parametro, e sia per uno specifico, creando quindi una funzione esclusiva che estende un determinato widget rendendola utilizzabile solo da esso.

## ● Retrofit

**Retrofit** è una libreria utilizzata per eseguire chiamate **REST** verso un determinato server per ricevere informazioni in un formato comprensibile e gestibile dall’applicazione in uso, chiamato **POJO**<sup>5</sup>.

Più precisamente, per chiamata REST<sup>6</sup> si intende uno stile architettonico software, cioè l’insieme delle decisioni significative sull’organizzazione dei sistemi software, per i sistemi distribuiti basato su **HTTP**<sup>7</sup>. Il funzionamento prevede una struttura degli **URL**<sup>8</sup> ben definita che identifica univocamente una risorsa o un insieme di risorse e l’utilizzo dei verbi HTTP specifici per il recupero di informazioni, utilizzato in questo programma, per la modifica e per altri scopi.

Per accedere a tali risorse, però, vi è la necessità di fornire un’informazione specifica su cui eseguire una ricerca così da ottenere dati chiave, denominata **URI**<sup>9</sup>. Questa informazione può essere passata attraverso l’URL al quale si esegue la chiamata al server per avere come restituzione una collezione di dati da gestire all’interno dell’applicativo e può essere, ad esempio, un ISBN di un libro, un indirizzo di posta elettronica, un file, ecc....

Il formato restituito da tale chiamata non è unico, ma, al contrario, può averne molteplici, come ad esempio JSON, XML o HTTP, ma anche file o immagini.

---

<sup>5</sup> Plain Old Java Object

<sup>6</sup> Representational State Transfer

<sup>7</sup> Hypertext Transfer Protocol

<sup>8</sup> Uniform Resource Locator

<sup>9</sup> Uniform Resource Identifier

**Retrofit** ragiona proprio su questo concetto, prendendo in considerazione un URL dato da un servizio specifico, esso è in grado di eseguire una chiamata per ricevere come risposta un possibile formato di dati che, nel caso di questo applicativo, vengono riconosciuti in **JSON** e **XML**.

Per eseguirne il corretto funzionamento però, è necessario implementare quanto segue:

- Avere bene a mente il tipo di URL al quale si voglia eseguire la ricerca e di dove inserire l'URI desiderato.
- Un Converter per deserializzare i dati ricevuti in base al formato della risposta.  
Per questo progetto in particolare verranno utilizzati due tipi di converter:
  - ❑ **GsonConverter** → Deserializzazione dei dati in formato JSON
  - ❑ **SimpleXMLConverter** → Deserializzazione dei dati in formato XML
- Un Builder al quale associare la base dell'URL dove si trova il servizio per la richiesta specifica.
- Un'interfaccia associata all'istanza della chiamata Retrofit che contiene al suo interno una funzione GET per ricevere i dati avente gli eventuali URI come parametri, annotati `@Query`.

## • **Glide**

**Glide** è una libreria che permette, prendendo in considerazione un URL, di eseguire una chiamata asincrona, quindi su thread separati, per ricevere immagini, video e GIF animate, applicando il risultato all'interno di un widget.

Può essere utilizzata anche in semplici modi, come caricare il contenuto e posizionarlo successivamente, oppure decidere in che modo convertire l'immagine, se Drawable o Bitmap, dove posizionarla o tagliarla all'interno del widget, se centrata, scontornata nei margini, zoomata o inserita in modo circolare, aggiungere dei placeholder durante il caricamento o nel caso in cui si vada incontro ad un errore, oppure modificare la dimensione in pixel del file multimediale, mantenendo però l'aspect ratio.

Una particolarità in più di questa libreria è la possibilità di migliorare la qualità d'immagine, ricevuta anche prima della visualizzazione, e di poterla salvare all'interno della cache nel momento in cui questa viene prelevata, cioè quando viene eseguita la chiamata. Glide, a quel punto, verificherà che l'URL ricevuto sia corretto e che restituisca uno dei formati precedentemente descritti, dopodichè salverà quel risultato per poterlo reperire in caso venga eseguita una chiamata successiva avente lo stesso URL, senza la necessità di avere una connessione ad internet attiva.

## • **RecyclerView**

Lista dinamica, ne verrà descritto il funzionamento in seguito.

## 2.2.2 Funzionamento

Questa schermata si occupa di ricevere dall’utente uno specifico dato tra ISBN, titolo o nome dell’autore di un determinato libro.

Al momento dell’inserimento, in base alla propria scelta, verranno visualizzati uno o più risultati inerenti al dato inserito, sotto forma di immagine e servizio al quale è collegata. Più dettagliatamente, all’avvio dell’applicazione, il programma caricherà inizialmente una custom toolbar per dare più risalto al proprio titolo e un Option Menu, cioè un menù a scomparsa posizionato in alto a destra, che permette di selezionare il tipo dei dati da cercare oppure passare alla visualizzazione della schermata inerente alla History, di cui se ne discuterà in seguito.

Successivamente, sarà avviata l’**Activity**<sup>10</sup> principale, cioè il contenitore dove stanziare tutte le destinazioni, cioè i **Fragment**<sup>11</sup>, all’interno del Grafico di Navigazione, che avrà il compito di gestire e visualizzare le schermate specifiche in base alle scelte effettuate dall’utente.

In questo caso, verrà avviata la destinazione principale, incaricata di eseguire la ricerca. A questo punto l’utente è invitato a inserire un dato specifico all’interno di una casella di testo per poter avviare la ricerca principale, questa effettuata utilizzando due servizi differenti, quali **Goodreads** e **Google Books**.

Nel caso in cui l’utente inserisca un ISBN, questo verrà prima verificato che sia valido, per poi ricercare all’interno del servizio Goodreads il risultato, o i risultati, che possiedano almeno l’immagine di copertina, così da poterla visualizzare, oppure, nel caso in cui non ve ne siano, verrà effettuato un sistema di fallback verso Google Books, cioè verrà terminata la chiamata **Retrofit** eseguita su Goodreads ed avviata una nuova successiva tramite il servizio Google Books.

Nel caso in cui venga inserito il nome del libro o dell’autore, invece, il servizio di riferimento sarà esclusivamente Google Books, dato che Goodreads non possiede le informazioni necessarie per poter eseguire tale ricerca.

Nel caso in cui non venisse trovato alcun riferimento al dato inserito, l’utente verrà informato con un messaggio a schermo, altrimenti verranno visualizzate tutte le immagini collegate con il proprio servizio tramite una **RecyclerView**, cioè una lista di layout fissi riempita dinamicamente in base al numero di elementi passati, e con tutte le informazioni inerenti alla stessa salvate in uno spazio apposito, come ISBN, titolo, autori, servizio e data di pubblicazione. Ognuna di queste è interagibile, in grado di passare le informazioni descritte alla destinazione successiva per decidere quale azione intraprendere con la cover selezionata. Nel caso in cui non si volesse inserire manualmente il codice, l’utente può passare al **Barcode Scanner** per reperire l’ISBN di un libro tramite la fotocamera.

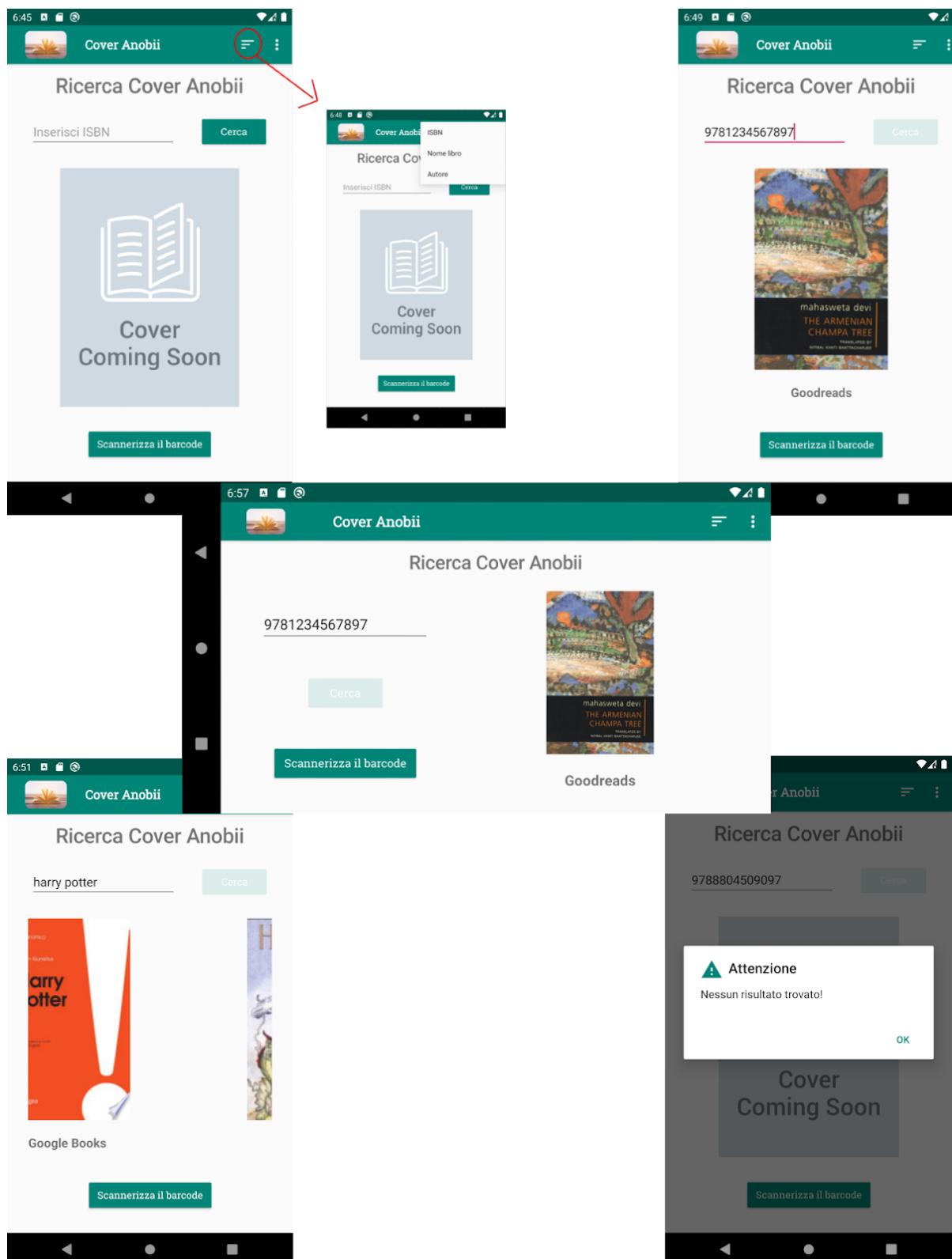
Queste chiamate vengono definite all’interno della classe **ViewModel** collegata alla schermata stessa, così da modificarne la UI in contemporanea alla risposta Retrofit.

---

<sup>10</sup> Finestra che contiene l’interfaccia utente

<sup>11</sup> Sub-activity, con un proprio scopo funzionale e ciclo di vita

### 2.2.3 Esempi Layout



## 2.2.4 Esempi Codice

```

private const val BASEURL GOOGLE = "https://www.googleapis.com/books/v1"
private const val BASEURL GOODREADS = "https://www.goodreads.com/search/"

// Build per la chiamata retrofit a Google Books con JSON converter
private val retrofitGoogle = Retrofit.Builder().baseUrl(BASEURL GOOGLE)
    .addConverterFactory(GsonConverterFactory.create())
    .build()

// Build per la chiamata retrofit a Goodreads con XML Converter
private val retrofitGoodreads = Retrofit.Builder().baseUrl(BASEURL GOODREADS)
    .addConverterFactory(SimpleXmlConverterFactory.create())
    .addConverterFactory(AnnotationStrategy())
    .build()

// Interfaccia utilizzata per ricevere un risultato della chiamata
interface GoogleBooks {
    @GET("volumes") // Preleva il risultato richiamando la funzione
    fun getBooks(
        @Query("q") isbn: String // Ulteriore dato da inserire per ricevere il giusto risultato
    ): Call<Books>
    // Valore restituito tramite un oggetto Call dello stesso tipo della classe di decodifica
}

interface GoodReadsBooks {
    @GET("index.xml")
    fun getBooks(
        @Query("q") isbn: String,
        @Query("key") key: String
    ): Call<GoodreadsResponse>
}

// Oggetto usato per instanziare la chiamata retrofit utilizzando la rispettiva interfaccia
object GoogleBooksApi {
    val retrofitService: GoogleBooks = retrofitGoogle.create(GoogleBooks::class.java)
}

object GoodReadsApi {
    val retrofitService: GoodReadsBooks = retrofitGoodreads.create(GoodReadsBooks::class.java)
}

```

```

/*
 * Load Image
 * Funzione utilizzata esclusivamente dalle ImageView, permette di inserire un'immagine specifica all'interno delle stesse quando si esegue una ricerca.
 * Nel caso in cui non ci siano risultati, cioè null, viene applicato un placeholder.
 * Nel caso in cui non ci sia connessione, cioè connection_error, viene applicata una notifica di connessione inesistente.
 * Nel caso in cui vengono trovati dei risultati, la libreria Glide eseguirà una ricerca tramite l'url per ricavare un drawable da inserire nell'ImageView
 * L'annotazione @BindingAdapter eseguite a questa funzione di assert chiamata all'interno di un layout che utilizza il Binding
 * #nircam url url all'immagine da visualizzare
 */

@BindingAdapter("loadImage")
fun ImageView.loadImage(url: String?) {
    apply { this.setImageResource(R.drawable.book_placeholder) }
    when (url) {
        null -> setImageResource(R.drawable.ic_connection_error)
        "connection_error" -> setImageResource(R.drawable.ic_connection_error)
        "results_ok" -> {}
        else -> {
            Glide.with(context).asDrawable().load(url)
                .apply(
                    RequestOptions()
                        .fitCenter()
                        .format(DecodeFormat.PREFER_ARGB_8888)
                        .override(Target.SIZE_ORIGINAL)
                        .placeholder(circularProgressDrawable)
                        .error(R.drawable.ic_broken_image)
                )
                .into(this)
        }
    }
}

```

```

<?xml version="1.0" encoding="utf-8"?>
<layout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto" >

    <data>
        <import type="android.view.View" />
        <variable
            name="viewModel"
            type="com.ovolab.coveranobii.cover.CoverViewModel" />
    </data>

    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".cover.CoverFragment">

        <TextView
            android:id="@+id/title"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginTop="10dp"
            android:text="Ricerca Cover Anobii"
            android:textAppearance="@style/TextAppearance.Title"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent" />
    
```

```

    fun getImageGoodreads(isbn: String, key: String) {
        val getBooks = GoodReadsApi.retrofitService.getBooks(isbn, key)

        // Placeholder caricato avviato
        _progressVisible.value = true
        _imageVisible.value = null

        getBooks.enqueue(object : Callback<GoodreadsResponse> {
            override fun onFailure(call: Call<GoodreadsResponse>, t: Throwable) {
                _progressVisible.value = false
                _imageVisible.value = connection_error
                _visualizing.value = true
            }

            override fun onResponse(call: Call<GoodreadsResponse>, response: Response<GoodreadsResponse>) {
                val listBooks = response.body()

                listBooks?.let {
                    // Nel caso non ci siano risultati, si eseguirà la chiamata successiva
                    if (it.search.totalResults!! > 0) {
                        val data = arraylistOf<InfoBooks>()

                        // La risposta deve contenere informazioni riguardanti i libri
                        it.search.results?.let { listwork -
                            for (book in listwork) {
                                // La risposta deve contenere un'immagine
                                book.bookCover?.let { image -
                                    if (image.contentName == "nophoto", ignoreCase = true) {
                                        data.add(
                                            InfoBooks(
                                                name = "Goodreads",
                                                isbn = book.isbn,
                                                title = book.title,
                                                author = book.author,
                                                year = book.originalPublicationYear?.toString(),
                                                img: "no_yeah",
                                                image.replace(
                                                    oldValue = "5000",
                                                    newValue: "5X500"
                                                )
                                            )
                                    }
                                }
                            }
                        }
                    }
                }
            }
        })
    }

    // Se sono stati trovati dei risultati, questi verranno salvati e visualizzati, altrimenti si eseguirà la chiamata successiva
    if (_data.isNotEmpty()) {
        _image.value = data
        _progressVisible.value = false
        _visualizing.value = true
        _imageVisible.value = results_ok
    } else {
        getImageGoogle(isbn)
    }
} else {
    getImageGoogle(isbn)
}

```

## 2.3 *Barcode Scanner*

### 2.3.1 *Requisiti*

- **ViewModel e LiveData**

Vedi [2.2.1 Requisiti](#)

- **Dexter e i Permessi**

I **Permessi** in Android vengono introdotti a partire dalla versione 6.0, definiti come autorizzazioni che l'utente deve concedere affinché le funzionalità della schermata corrente vengano eseguite correttamente.

Queste devono essere stilate all'interno di una risorsa XML chiamata **Android Manifest**, cioè dove sono salvate tutte le informazioni necessarie ad eseguire ogni porzione dell'applicativo, come il numero di classi, il nome del package, le librerie, i servizi, ecc..., e possono essere tutte di diverse categorie, come per la vibrazione, l'uso della fotocamera, internet o per la lettura e scrittura della memoria principale. Alcune schermate di questa applicazione necessitano di autorizzazioni da parte dell'utente per gestire alcune funzionalità, la stesura del codice per permettere questa procedura, però, può risultare tediosa a lungo andare e con un dispendio di righe non indifferente.

Arriva quindi in soccorso **Dexter**, una libreria sviluppata da uno studio di programmazione chiamato Karumi, che permette di gestire le richieste di autorizzazioni in modo rapido e stilizzato.

Per eseguirne un corretto funzionamento, vi è il bisogno di implementare quanto segue:

- **Contesto** al quale è collegata la schermata
- Lista dei **permessi** al quale si vuole dare accesso
- **Listener** per gestire la visualizzazione e le scelte di concessione che esegue l'utente, nei casi in cui decida se:

- Concedere i permessi e continuare con la prosecuzione
- Negare i permessi e tornare alla schermata precedente
- Negare permanentemente i permessi così da inviare l'utente verso le impostazioni del dispositivo per abilitare manualmente le autorizzazioni necessarie

Questi possono essere di due diverse categorie:

- **PermissionListener** per la gestione dell'autorizzazione di un singolo permesso
- **MultiplePermissionsListener** per l'autorizzazione di più permessi

## ● Google Play Services Vision e API

La suite di **Google Mobile Vision**, introdotta con il rilascio di Google Play Service 7.8, si può riassumere in quelle che si definiscono **API**<sup>12</sup>, cioè un set di definizioni e protocolli con i quali vengono realizzati ed integrati software applicativi, consentendo all'applicazione di comunicare con altri prodotti o servizi senza sapere come questi vengano implementati, semplificando così lo sviluppo generale. Grazie alle API è possibile collegare con facilità una specifica infrastruttura mediante lo sviluppo di app-cloud native, nonché condividere dati con altri utenti esterni.

Esistono tre diversi approcci ai criteri di rilascio delle API:

- **Privato** → L'API è destinata solo ad un utilizzo interno, approccio che offre alle aziende un controllo ottimale.
- **Partner** → L'API è condivisa tra specifici partner aziendali, approccio che può fornire flussi di reddito aggiuntivi, senza compromettere la qualità.
- **Pubblico** → L'API è disponibile a chiunque, approccio che consente a terze parti di sviluppare app che interagiscono con l'API e che possono rappresentare una fonte di innovazione.

Queste risultano essere estremamente preziose poiché oltre a semplificare ed espandere il modo in cui un'azienda si collega ai propri partner, permettono anche di monetizzare i dati, basti pensare alle API di Google Maps.

Mobile Vision è quindi una API pubblica, facente parte del **Kit ML**<sup>13</sup>, cioè una collezione di tecniche di programmazione che offre alle app la possibilità di apprendere e migliorare automaticamente dall'esperienza senza essere esplicitamente programmato per farlo, per la ricerca di oggetti in foto e video e che include dei rilevatori, incaricati di localizzare e descrivere oggetti visivi in immagini o frame video.

Per il funzionamento di questa schermata, però, verrà utilizzata solo una di essi, cioè **Barcode API**, un'interfaccia specifica utilizzata per rilevare e scannerizzare i codici a barre a 1D, come quelli presenti sul retro di un libro, e a 2D, come i QR-Code, in tempo reale, sul dispositivo e con qualsiasi orientamento.

---

<sup>12</sup> Application Programming Interface

<sup>13</sup> Machine Learning

### 2.3.2 Funzionamento

Nel caso l'utente decida di voler passare ad una lettura del codice a barre presente sul retro di un libro, premendo il pulsante descritto nella schermata precedente, verrà reindirizzato in una nuova schermata.

La sezione dell'applicazione in questione consente di scannerizzare, utilizzando la **fotocamera** posteriore del dispositivo, il codice a barre univoco di un libro ottenendo il relativo ISBN, ma andiamo con ordine.

All'avvio di essa, gestito dalla libreria **Dexter**, l'utente avrà la possibilità di scegliere se autorizzare o meno i permessi inerenti all'utilizzo della fotocamera.

Nel caso di una risposta positiva, dato che le alternative negative sono state precedentemente elencate nel paragrafo [2.3.1 Requisiti](#), l'utente vedrà apparire sullo schermo la rappresentazione di cosa la fotocamera posteriore sta puntando.

Questo è reso possibile grazie ad una determinata View, cioè il widget base per la creazione di elementi dell'interfaccia utente, chiamata **SurfaceView**.

Questo widget, non dilungandosi troppo sui particolari, permette una visualizzazione della schermata attuale come un gruppo di frame che si aggiornano regolarmente in modo gerarchico su un Thread, cioè un processo, differente da quello principale, come se fossero tutti elencati tramite un asse Z con il proprio turno di visualizzazione.

Al movimento della fotocamera, perciò, il frame precedentemente catturato da questa View verrebbe rimpiazzato con quello successivo, così da rendere l'idea di utilizzare effettivamente il servizio dell'immagine.

A questo punto, è possibile visionare l'ambiente circostante utilizzando il proprio dispositivo, bisogna però implementare la scannerizzazione dell'ambiente per poter identificare elementi di rilevo necessari alla continuazione.

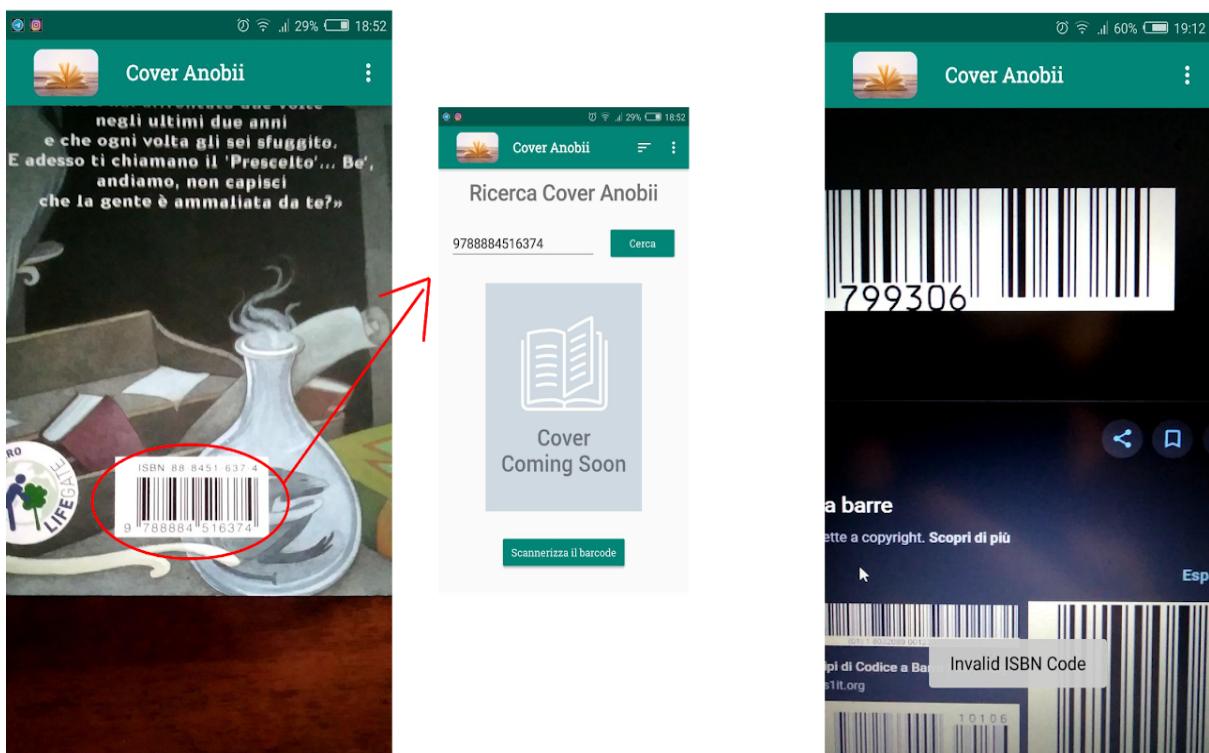
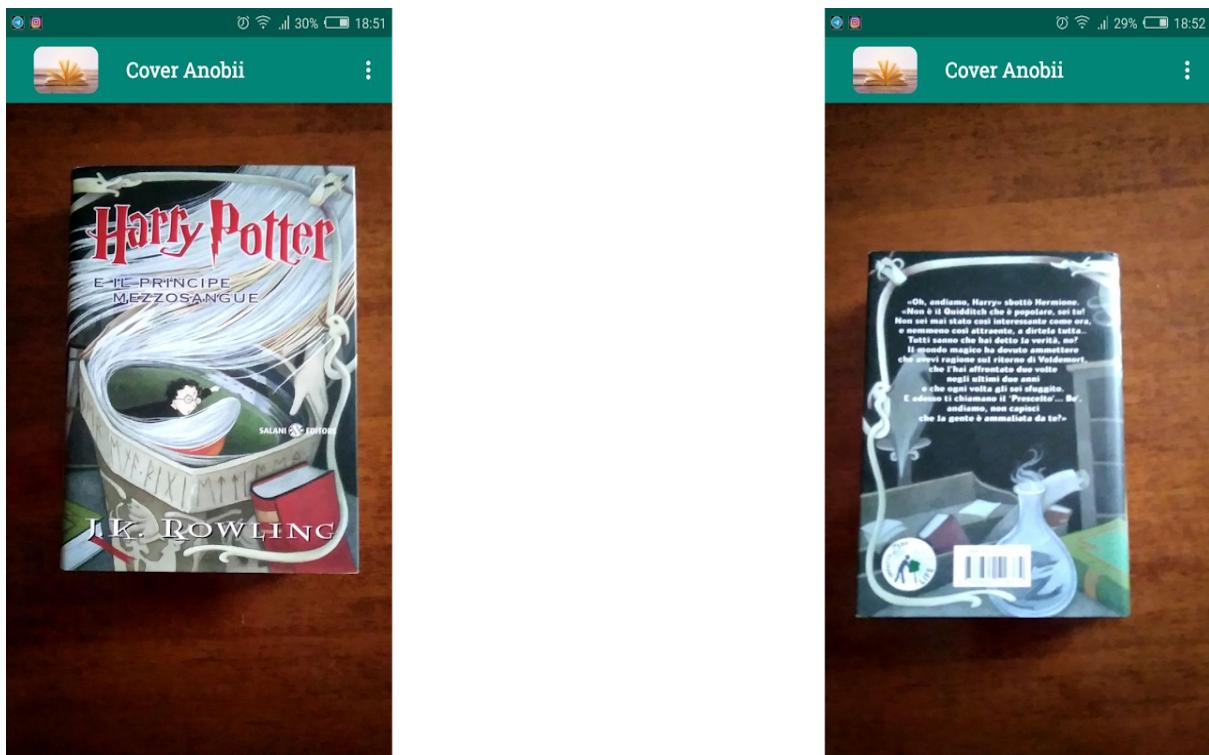
Innanzitutto bisogna riconoscere, tramite un **Barcode Detector**, quali tipi di barcode devono essere rilevati, per poi successivamente passare queste informazioni all'implementazione della fotocamera posteriore, in modo tale che vengano riconosciuti solo gli elementi essenziali alla prosecuzione.

Dopodichè bisogna definire l'**Holder** collegato alla SurfaceView, si tratta di un "contenitore" al quale allocare il processo da seguire per visualizzare e gestire il contenuto all'interno della suddetta View, perciò al momento della creazione, cioè dati i permessi, verrà avviata la fotocamera e al momento della distruzione, cioè all'uscita dalla schermata, verrà invece interrotta la visualizzazione.

Rimane solo un ultimo punto, la gestione della cattura del codice a barre.

Qui l'utente si ritroverà a scansionare un codice per rilevare l'ISBN desiderato facendo avviare un evento che, nel caso in cui non sia già stato scansionato in precedenza, procederà ad eseguire un leggero suono acustico e una vibrazione, per informare l'utente dell'avvenuta lettura, per poi salvare il codice catturato all'interno di una variabile **ViewModel**, incaricata di verificare prima che l'ISBN sia valido e successivamente di navigare verso la schermata di ricerca principale con tale informazione.

### 2.3.3 Esempi Layout



## 2.3.4 Esempi Codice

```

@Override fun onActivityCreated(savedInstanceState: Bundle?) {
    super.onActivityCreated(savedInstanceState)

    // Per poter eseguire un Bip acustico
    tone = ToneGenerator(AudioManager.STREAM_MUSIC, volume: 100)

    // Per poter eseguire la vibrazione
    vibrate = requireContext().getSystemService(Context.VIBRATOR_SERVICE) as Vibrator

    cameraView.visibility = View.GONE

    checkPermission()

    /*
     * Nel caso in cui l'ISBN letto da barcode sia valido, verrà passato a CoverFragment tramite
     * navigation controller
     */
    viewModel.isbnBarcode.observe(viewLifecycleOwner, Observer { it: String?
        it?.let { it: String
            if (isValidISBN(it)) {
                val arg = Bundle()
                arg.putString("isbnBarcode", it)

                this.findNavController().navigate(
                    R.id.action_barcodeFragment_to_coverFragment, arg, navOptions: null
                )

                viewModel.doneNavigation()
            } else {
                createToast(requireContext(), text: "Invalid ISBN Code")
            }
        }
    })
}
}

```

```

/***
 * Check Permission
 * Funzione utilizzata per gestire i permessi inerenti alla fotocamera con l'aiuto della
 * libreria Dexter.
 * Nel caso di permessi concessi, l'utente verrà indirizzato verso la prosecuzione del programma.
 * Nel caso di negati permanentemente, l'utente verrà indirizzato nelle impostazioni
 * dell'applicazione per la concessione manuale dei permessi
 */
private fun checkPermission() {
    Dexter.withActivity(activity).withPermission(Manifest.permission.CAMERA)
        .withListener(object: PermissionListener {
            override fun onPermissionGranted(response: PermissionGrantedResponse?) {
                cameraView.visibility = View.VISIBLE
                initialDetection()
            }

            override fun onPermissionRationaleShouldBeShown(permission: PermissionRequest?,
                token: PermissionToken?) {
                token?.continuePermissionRequest()
            }

            override fun onPermissionDenied(response: PermissionDeniedResponse?) {
                response?.let { it: PermissionDeniedResponse
                    if (it.isPermanentlyDenied) {
                        val intent = Intent(
                            Settings.ACTION_APPLICATION_DETAILS_SETTINGS,
                            Uri.fromParts("package", activity?.packageName, fragment: null))
                        intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK)
                        startActivity(intent)
                    }
                }
            }
        })
        .check()
}

```

```

/***
 * Initial Detection
 * Funzione utilizzata per visualizzare l'immagine visibile in fotocamera tramite una
 * SurfaceView, cioè una View che ragiona attraverso un Thread differente rispetto alla GUI
 */
private fun initialDetection() {

    // Istanza per identificare il barcode
    val barcodeDetector = BarcodeDetector.Builder(context)
        .setBarcodeFormats(Barcode.ALL_FORMATS).build()

    // Istanza per abilitare la fotocamera posteriore
    cameraSource = CameraSource.Builder(context, barcodeDetector)
        .setRequestedPreviewSize(1920, 1080).setAutoFocusEnabled(true)
        .setFacing(CameraSource.CAMERA_FACING_BACK)
        .setRequestedFps(15.0f)
        .build()

    // SurfaceView Holder, utilizzabile per determinare le operazioni da eseguire quando il
    // thread viene creato o distrutto
    cameraView.holder.addCallback(object: SurfaceHolder.Callback {
        override fun surfaceChanged(holder: SurfaceHolder?, format: Int, width: Int,
            height: Int) {}

        override fun surfaceDestroyed(holder: SurfaceHolder?) {
            cameraSource.stop()
        }
    })

    @SuppressLint("MissingPermission")
    override fun surfaceCreated(holder: SurfaceHolder?) {
        // Viene avviata la visualizzazione della fotocamera all'interno della view
        cameraSource.start(cameraView.holder)
    }
}

```

```

var preValue = ""

// Viene creato un processo per identificare il barcode visualizzato tramite fotocamera
barcodeDetector.setProcessor(object: Detector.Processor<Barcode> {
    override fun release() {}

    override fun receiveDetections(detect: Detector.Detections<Barcode>?) {
        val barcodes = detect?.detectedItems

        /*
         * Nel caso in cui sia stato trovato un barcode e non sia già stato scannerizzato in
         * precedenza, quel valore viene salvato per poi essere controllato, facendo partire
         * un bip acustico ed una vibrazione
         */
        if (barcodes!!.size() != 0 && preValue != barcodes.valueAt(index: 0).displayValue) {
            Handler(Looper.getMainLooper()).post {
                preValue = barcodes.valueAt(index: 0).displayValue
                viewModel.setISBNBarcode(preValue)
                tone.startTone(ToneGenerator.TONE_CDMA_PIP, durationMs: 150)
                if (Build.VERSION.SDK_INT > Build.VERSION_CODES.O) {
                    vibrate.vibrate(VibrationEffect.createOneShot(milliseconds: 150,
                        VibrationEffect.DEFAULT_AMPLITUDE))
                } else {
                    @SuppressLint("DEPRECATION")
                    vibrate.vibrate(milliseconds: 150)
                }
            }
        }
    }
})
}

```

## 2.4 *Image Management*

### 2.4.1 *Requisiti*

- **ViewModel e LiveData**

Vedi [2.2.1 Requisiti](#)

- **Data Binding**

Vedi [2.2.1 Requisiti](#)

- **Glide**

Vedi [2.2.1 Requisiti](#)

- **Dexter e i Permessi**

Vedi [2.3.1 Requisiti](#)

- **Firebase Firestore e Storage**

**Firebase** è un potente servizio on-line che permette di salvare e sincronizzare i dati elaborati da applicazioni web e mobile.

Si tratta di un database **NoSQL<sup>14</sup>**, cioè un sistema software dove la persistenza dei dati è in generale caratterizzata dal fatto di non utilizzare il modello relazionale, di solito usato dalle basi di dati tradizionali, come ad esempio il RDBMS<sup>15</sup>.

L'espressione fa riferimento al linguaggio SQL<sup>16</sup>, che è il più comune linguaggio di interrogazione dei dati nei database relazionali, qui preso a simbolo dell'intero paradigma relazionale.

Firebase rientra quindi in una categoria di servizi on-line in rapidissima diffusione e chiamati **backend**, questi consentono l'interagibilità con servizi remoti, tramite l'utilizzo di API, per ricevere e scambiare informazioni con gli utenti in Cloud.

Queste informazioni vengono gestite quindi da un servizio denominato **Cloud Firestore**, facente parte del pacchetto Firebase.

Cloud Firestore è un database flessibile e scalabile per lo sviluppo di dispositivi Web e Mobile, mantenendo i propri dati sincronizzati tra le app client attraverso listener in tempo reale, offrendo un'ulteriore supporto offline in modo da poter creare app reattive che funzionino indipendentemente dalla latenza della rete o dalla connettività Internet.

---

<sup>14</sup> Non-SQL, non-relazionale

<sup>15</sup> Relational Database Management System

<sup>16</sup> Structured Query Language

Le funzionalità chiave principali di questo servizio sono le seguenti:

- **Flessibilità** → Documenti nidificati.
- **Interrogazione espressiva** → Filtraggio e reperimento di dati tramite query.
- **Aggiornamenti in tempo reale** → Sincronizzazione ed aggiornamento dei dati su qualsiasi dispositivo collegato.
- **Supporto offline** → Memorizzazione in cache dei dati che l'applicazione sta attivamente utilizzando.
- **Adattamento** → Utilizzo della potente infrastruttura di Google Cloud Platform.

Il funzionamento di tale servizio non è molto diverso da quello di una normale gestione di un database, cioè con modalità di selezione, organizzazione, eliminazione ed aggiornamento, tranne che vengono introdotte alcune nozioni logiche, cioè, in questo caso, i database vengono definiti come **raccölte**, o **collezioni**, che possono contenere uno o più **documenti**, cioè tabelle, definendo al loro interno i dati relativi a quella specifica tabella, cioè i **campi**.

È inoltre possibile creare raccolte secondarie all'interno di documenti, creando così strutture di dati gerarchiche che si adattano al crescere del database.

Oltre al servizio di archiviazione dati, vi è uno analogo per il salvataggio di materiale multimediale e non, chiamato **Cloud Storage**.

Questo risulta essere un servizio di archiviazione potente, semplice ed economico. Gli **SDK**<sup>17</sup> Firebase per l'archiviazione su cloud aggiungono la sicurezza di Google ai file caricati e scaricati per le proprie app Firebase, indipendentemente dalla qualità della rete, utilizzati per archiviare immagini, audio, video o altri contenuti generati dagli utenti.

Le funzionalità chiave principali di questo servizio sono le seguenti:

- **Operazioni robuste** → Esecuzione di upload e download indipendentemente dalla qualità della rete.
- **Forte sicurezza** → Integrazione con l'autenticazione Firebase.
- **Alta scalabilità** → Progettato per scalare exabyte quando l'app diventa virale.

Il funzionamento di tale servizio è relativamente basilare, questo infatti archivia i file in un bucket, cioè uno spazio dedicato, rendendoli accessibili sia tramite Firebase che Google Cloud. Ciò offre la flessibilità di caricare e scaricare file da client mobili tramite gli SDK di Firebase e di eseguire elaborazioni sul lato server come il filtro di immagini o la transcodifica video utilizzando Google Cloud Platform.

---

<sup>17</sup> Software Development Kit

## 2.4.2 Funzionamento

L'utente ha così scelto la cover perfetta da suggerire durante la ricerca precedente, premendo quindi su tale immagine verrà trasportato, assieme alle informazioni inerenti alla suddetta, alla sezione successiva, chiamata **Image Management**.

All'interno di questa schermata, l'utente potrà, oltre a visualizzare l'immagine caricata tramite **Glide**, eseguire tutte le operazioni relative a tale immagine come salvataggio e suggerimento sul servizio dedicato.

Ma andiamo con ordine, le operazioni principali sono le seguenti:

- **Salvataggio e reperimento immagine**

Qui possono essere eseguite le operazioni di salvataggio e reperimento immagine all'interno della galleria del dispositivo, esse però sono disponibili solo se l'utente acconsente la scrittura e lettura della memoria tramite **Dexter**.

Alla pressione del pulsante di salvataggio, verranno prelevate le informazioni come il titolo e l'estensione dell'immagine, cioè JPG, per poi salvarne una copia in una sezione apposita della galleria utilizzando due librerie specifiche per tale operazione, quali **contentResolver** e **MediaStore**.

Alla pressione del pulsante di reperimento, invece, verrà avviata un'Activity che visualizzerà tutte le immagini salvate all'interno del dispositivo, con l'intento di prelevarne una a scelta. Non appena l'utente avrà selezionato quella desiderata, egli verrà reindirizzato nella schermata precedente con l'immagine selezionata ben visibile.

- **Salvataggio e reperimento immagine in Cloud**

Qui possono essere eseguite le operazioni, selezionabili tramite il bottone situato nell'OptionMenu e controllando che l'accesso ad internet sia attivato, di salvataggio e reperimento immagine all'interno dello spazio **Cloud Storage**.

Alla pressione del pulsante di salvataggio, l'immagine visualizzata verrà salvata all'interno del bucket in una cartella dove vengono posizionate tutte le cover salvate, per essere utilizzate in un secondo momento, creando anche una collezione all'interno di **Firebase** contenente documenti aventi il nome dell'immagine salvata, in questo modo possono essere successivamente reperibili.

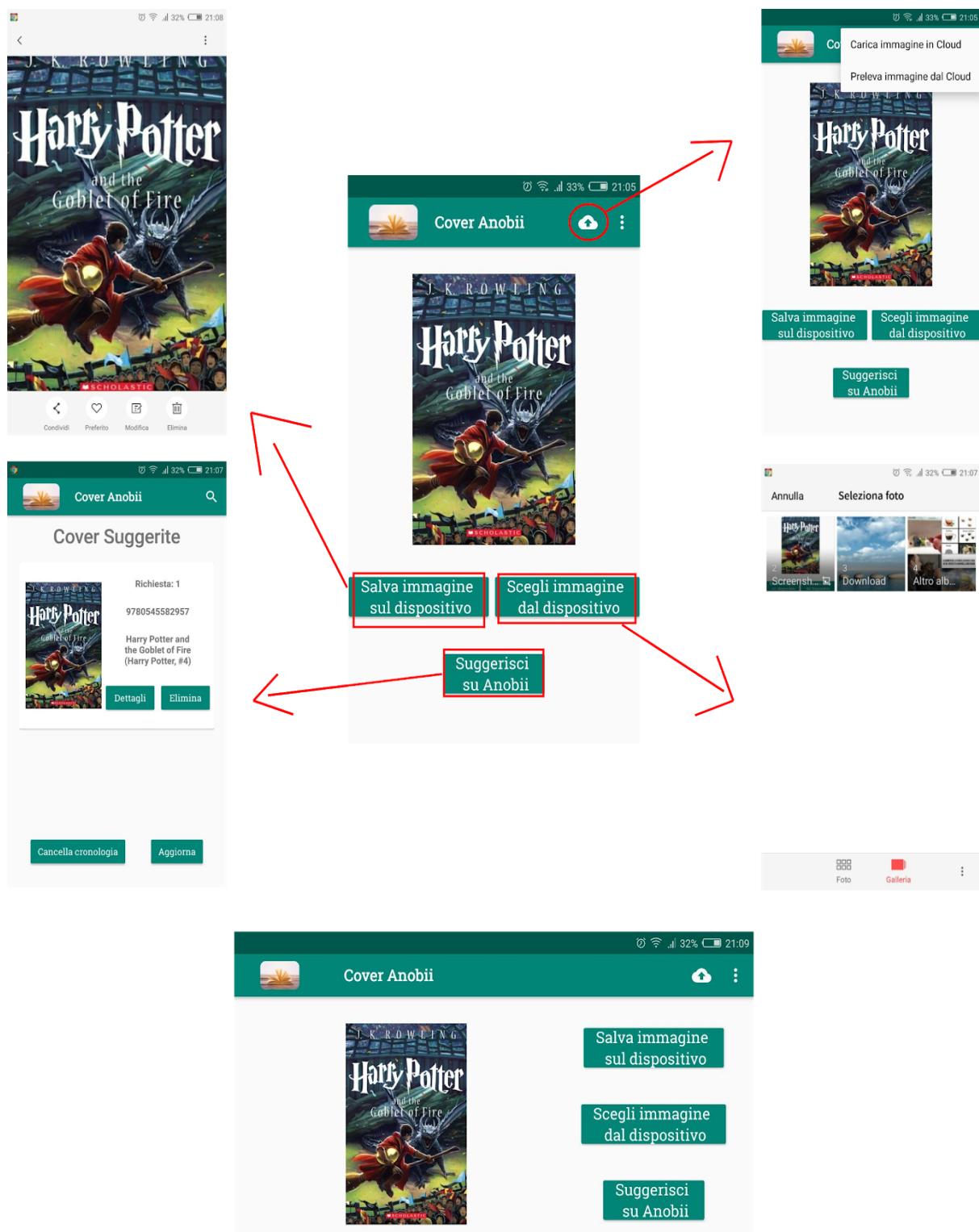
Alla pressione del pulsante di reperimento, invece, verrà avviata la schermata di **Cloud Images**, di cui ne verrà spiegato il funzionamento in seguito.

- **Suggerimento su Anobii**

Qui vengono suggerite le cover verso il sistema Anobii, tuttora solo in modo fittizio, con il successivo salvataggio di essa all'interno della cronologia suggerimenti, la quale verrà descritta in seguito.

Nel momento in cui l'utente deciderà di suggerire tale cover, verrà creata una collezione in **Firebase** chiamata "Books" con all'interno il numero di richiesta, come documento, e i dati della suddetta, come campi, oltre a salvarne una copia all'interno di **Cloud Storage**.

### 2.4.3 Esempi Layout



## 2.4.4 Esempi Codice

```


/*
 * Save Image To Internal
 * Salva l'immagine selezionata in CoverFragment all'interno della galleria del dispositivo
 */
private fun saveImageToInternal() {
    val externalStorageState = Environment.getExternalStorageState()

    if (externalStorageState == Environment.MEDIA_MOUNTED) {

        val resolver = requireContext().contentResolver
        val contentValues = ContentValues().apply { this.ContentValues()
            put(MediaStore.MediaColumns.DISPLAY_NAME, infoBook.title)
            put(MediaStore.MediaColumns.MIME_TYPE, "image/jpeg")
            if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.Q) {
                put(MediaStore.MediaColumns.RELATIVE_PATH, "DCIM/Books")
            }
        }

        val uri = resolver.insert(MediaStore.Images.Media.EXTERNAL_CONTENT_URI, contentValues)

        resolver.openOutputStream(uri!!).use { it: OutputStream?
            val drawable = binding.selectedImage.drawable.toBitmap()
            drawable.compress(Bitmap.CompressFormat.JPEG, quality: 100, it)
            it!!.flush()
            it.close()
            createToast(requireContext(), text: "Image save success")
        }

    } else {
        createToast(requireContext(), text: "Unable to access to storage")
    }
}


```

```


/**
 * Pick Image
 * Funzione che apre una finestra di scelta inerente a quale immagine si desidera prelevare
 * dalla galleria per essere poi visualizzata nella sezione apposita
 */
private fun pickImage() {
    val intent = Intent(Intent.ACTION_PICK)
    intent.type = "image/*"
    startActivityForResult(intent, IMAGE_PICK_CODE)
}

override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {

    // Nel caso in cui venga selezionata un'immagine, l'ImageView avrà la stessa come contenuto
    if (resultCode == Activity.RESULT_OK && requestCode == IMAGE_PICK_CODE) {
        binding.selectedImage.setImageURI(data?.data)
    }
}

override fun onPause() {
    super.onPause()

    // Per evitare di inviare più e più volte la stessa immagine attualmente visibile
    sameImage = false
}


```

```


/**
 * Add Image To Suggest
 * Funzione che verifica quale sia l'ultimo elemento inserito, aggiungendone
 * uno alla posizione successiva
 */
private fun addImageToSuggest() {
    InitFirebase.mFirestore.collection( collectionPath: "Books").get()
        .addOnSuccessListener { listDoc ->
            var i = 1

            if (!listDoc.isEmpty) {
                if (listDoc.documents.isNotEmpty()) {
                    i = listDoc.documents.last().id.toInt() + 1
                }
            }

            viewModel.addInfoImage(infoBook,
                requireActivity().application,
                binding.selectedImage.drawable.toBitmap(), i)
        }
}


```

```


/**
 * Add Info Image
 * Funzione che inserisce all'interno del database remoto e locale le informazioni della
 * cover suggerita, nonché l'immagine in Storage online
 * @param info informazioni della cover
 * @param application utilizzata per il funzionamento della Room
 * @param image immagine bitmap da inviare in Firebase Storage
 * @param request numero richiesta attribuito all'elemento
 */
fun addInfoImage(info: InfoBooks, application: Application, image: Bitmap, request: Int) {
    val name = "${UUID.randomUUID()}.jpg"
    val baos = ByteArrayOutputStream()
    val data: HashMap<String, String> = HashMap()

    image.compress(Bitmap.CompressFormat.JPEG, quality: 100, baos)
    val imageByte: ByteArray = baos.toByteArray()
    val imageCloudRef = mFirebaseStorage.reference.child(name)
    imageCloudRef.putBytes(imageByte)

    data["isbn"] = info.isbn
    data["service"] = info.service
    data["title"] = info.title
    data["author"] = info.author
    data["publicationYear"] = info.publicationYear
    data["image"] = name

    mFirestore.collection( collectionPath: "Books").document(request.toString()).set(data)
        .addOnFailureListener { exception:
            uiScope.launch { this: CoroutineScope
                InfoBooksDatabase.get(application).getInfoBooksDao.insertBook(
                    InfoBooksTable(request, info.isbn, info.service, info.title, info.author,
                        info.publicationYear, name)
                )
            }
        }
}


```

## 2.5 *Cloud Images*

### 2.5.1 *Requisiti*

- **ViewModel e LiveData**

Vedi [2.2.1 Requisiti](#)

- **Glide**

Vedi [2.2.1 Requisiti](#)

- **Firebase Firestore e Storage**

Vedi [2.4.1 Requisiti](#)

- **RecyclerView**

Precedentemente si è parlato di una **RecyclerView** come una lista dinamica di elementi, ma cerchiamo di spiegarne il funzionamento più nello specifico.

Le RecyclerView vengono introdotte a partire dalla versione Android 5.0 Lollipop come alternativa alla visualizzazione di liste statiche usando ListView, permettendo il settaggio di singoli elementi con la loro relativa creazione solo e soltanto nel caso in cui venga specificato il numero di essi da visualizzare.

Per eseguirne il corretto funzionamento vi è il bisogno di implementare alcuni componenti, quali:

- **Layout Manager** → È responsabile della creazione e del posizionamento delle view all'interno del RecyclerView. Esistono diverse tipologie di LayoutManager come il LinearLayoutManager, utilizzato per creare liste orizzontali o verticali, o GridLayoutManager, per creare, invece, griglie.
- **Data Source** → È l'insieme di dati utilizzato per popolare la lista tramite l'Adapter.
- **Adapter** → È responsabile di estrarre i dati dal Data Source e di usare questi dati per creare e popolare i ViewHolder, cioè i singoli elementi. Quest'ultimi saranno poi inviati al Layout Manager dell'Adapter.
- **View Holder** → È la chiave di volta tra il RecyclerView e l'Adapter e permette la riduzione nel numero di view da creare. Questo oggetto infatti fornisce il layout da popolare con i dati presenti nel DataSource e viene utilizzato dal RecyclerView per ridurre il numero di layout da creare per popolare la lista. Tali elementi possono non solo essere visualizzati, ma anche interagibili, potendo inserire bottoni oppure rendendo l'intero elemento collegato ad un'azione.

## 2.5.2 Funzionamento

Supponendo che, nella schermata precedente, l’utente desideri salvare la cover visualizzata all’interno di uno spazio dedicato in **Cloud**, così da essere gestita e resa reperibile anche da membri del proprio team o utenti esterni. Egli ha la possibilità di farlo, tramite l’utilizzo di questa schermata, avente come punti principali i servizi Firebase **Firestore** e **Storage**.

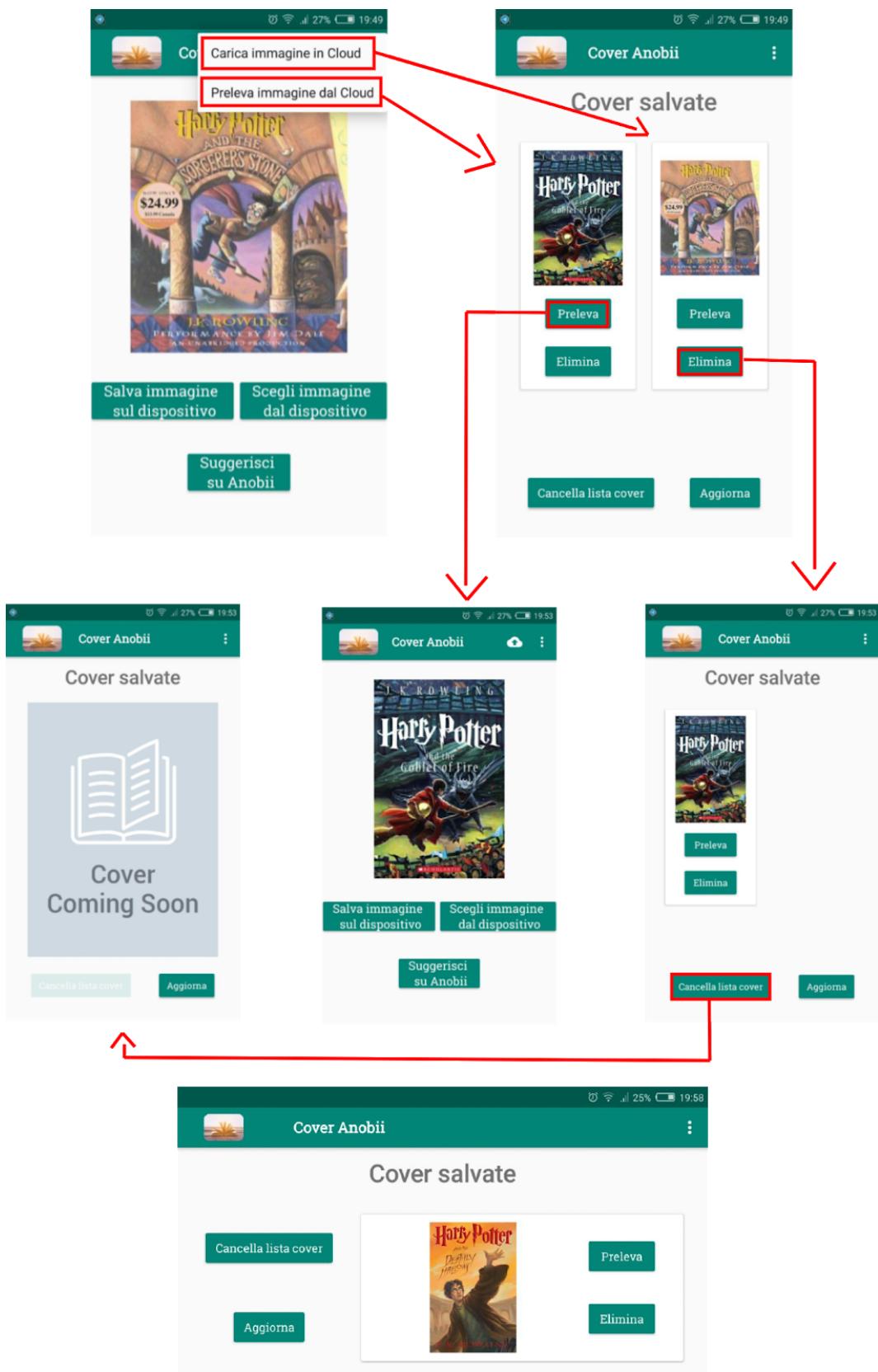
Più precisamente, alla pressione del pulsante inerente al salvataggio in Cloud, l’applicazione provvederà a estrapolare in formato **Bitmap** l’immagine visibile su schermo per poi salvare la suddetta in formato JPG all’interno di una cartella, situata nel bucket di Cloud Storage, chiamata “savedCover”, con un proprio codice univoco generato automaticamente, com’è già stato descritto nella sezione precedente.

Ora però si desidera prelevare un’immagine dal Cloud, alla pressione quindi del pulsante inerente al prelievo, l’utente verrà indirizzato verso la schermata successiva, cioè **Cloud Images**. Qui egli potrà visualizzare una lista contenente tutte le immagini salvate, caricate e visualizzate tramite **Glide**, come fossero all’interno di una griglia con la possibilità di prelevare od eliminare una qualsivoglia a scelta. Ogni elemento presente è inoltre posizionato all’interno di quella che viene definita **CardView**, cioè un widget, facente parte di **AndroidX**, che permette la visualizzazione dei suddetti come fossero all’interno di un quadro con bordi arrotondati e ombreggiatura per definire meglio la differenziazione. Esse vengono reperite tramite un sistema di matching, dove il nome dell’immagine salvata deve coincidere con quello situato all’interno della collezione Firestore inerente a questo tipo di cover, altrimenti l’applicazione non saprebbe quali immagini devono essere visualizzate e quali invece no.

Oltre alle normali operazioni utilizzabili verso il singolo elemento, vi sono quelle inerenti all’intera **RecyclerView**, come l’eliminazione di tutte le immagini salvate e l’aggiornamento di tutta la lista, nel caso in cui uno o più utenti inseriscono una o più immagini durante il proprio utilizzo dell’applicazione.

Al momento del reperimento, il programma preleverà l’immagine che avrà il matching con il documento Firestore selezionato, per renderla visibile all’interno dello spazio apposito nella schermata **Image Management**.

### 2.5.3 Esempi Layout



## 2.5.4 Esempi Codice

```
// Verrà visualizzato un placeholder nel caso in cui non ci siano cover salvate
viewModel.showPlaceholderNoImage.observe(viewLifecycleOwner, Observer { it:Boolean!
    if (it) {
        placeholderNoRequest.visibility = View.VISIBLE
        listCoverSent.visibility = View.GONE
    } else {
        placeholderNoRequest.visibility = View.GONE
        listCoverSent.visibility = View.VISIBLE
    }
    deleteAll.isEnabled = !it
})

// Verrà visualizzata una ProgressBar quando si effettuerà una chiamata in Cloud
viewModel.showProgressBar.observe(viewLifecycleOwner, Observer { it:Boolean!
    if (it) {
        progressHistory.visibility = View.VISIBLE
    } else {
        progressHistory.visibility = View.GONE
    }
})

// Elimina tutte le cover, controllando prima che vi sia una connessione internet attiva
deleteAll.setOnClickListener { it:View!
    if (checkConnection(requireContext())) {
        createConfirmAlert(requireContext(), title: "Attenzione",
            R.drawable.ic_baseline_warning_24,
            message: "Sei sicuro di voler eliminare tutte le cover salvate?",
            posText: "Elimina", DialogInterface.OnClickListener { _, _ ->
                viewModel.deleteAllBooks()
            })
    } else {
        createToast(requireContext(), text: "Connection Error")
    }
}

manageRecyclerCloud()
```

```
/*
 * Cloud Image Adapter
 * Classe adapter per la gestione dei singoli elementi all'interno della RecyclerView in
 * CloudImageFragment
 * Author Simone Tognetti - Ovalab
 */
class CloudImageAdapter(private val getCloudImageClickListener: GetCloudImageClickListener,
    private val deleteClickListener: DeleteClickListener): RecyclerView.Adapter<CloudImageAdapter.ViewHolder>() {

    // Dati da visualizzare
    var data = arrayListOf<String>()
        set(value) {
            field = value
            notifyDataSetChanged()
        }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
        return ViewHolder.from(parent)
    }

    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        val item = data[position]
        holder.bind(item, getCloudImageClickListener, deleteClickListener)
    }

    override fun getItemCount() = data.size
}
```

```
/*
 * Delete Image
 * Funzione che elimina un'immagine specifica sia da Firebase Firestore e sia da Storage per poi
 * ricaricare la lista di immagini salvate
 * @param name nome dell'immagine da eliminare
 */
fun deleteImage(name: String) {
    FirebaseFirestore.collection("collectionPath: \"savedCover\"").document(name).delete()
    FirebaseStorage.reference.child(pathString: "savedCover/$name.jpg").delete()
    getListImageCloud()
}

/**
 * Delete All Books
 * Funzione che elimina tutte le immagini salvate sia da Firebase Firestore e sia da Storage per
 * poi ricaricare la lista di immagini salvate
 */
fun deleteAllBooks() {
    FirebaseFirestore.collection("collectionPath: \"savedCover\"").get().addOnSuccessListener { listDoc ->
        if (listDoc.isEmpty()) {
            if (listDoc.documents.isNotEmpty()) {
                for (elem in listDoc.documents) {
                    FirebaseFirestore.collection("collectionPath: \"savedCover\"").document(elem.id).delete()
                        .addOnSuccessListener { it:Void!
                            FirebaseStorage.reference
                                .child(pathString: "savedCover/${elem.id}.jpg").delete()
                        }
                }
                getListImageCloud()
            }
        }
    }
}
```

```
/*
 * Add Image Cloud
 * Funzione che gestisce il salvataggio in Cloud delle "cover salvate" creandone un riferimento
 * in Firestore
 * @param image immagine da inviare in Cloud
 * @param context Context per creare un toast di avvenuto salvataggio
 */
fun addImageCloud(image: Bitmap, context: Context) {
    val name = UUID.randomUUID().toString()
    val baos = ByteArrayOutputStream()
    val data: HashMap<String, String> = HashMap()

    image.compress(Bitmap.CompressFormat.JPEG, quality: 100, baos)
    val imageByte: ByteArray = baos.toByteArray()
    val imageCloudRef = FirebaseStorage.reference.child(pathString: "savedCover/$name.jpg")
    imageCloudRef.putBytes(imageByte)

    FirebaseFirestore.collection("collectionPath: \"savedCover\"").document(name).set(data)
        .addOnSuccessListener { it:Void!
            createToast(context, text: "Immagine salvata in Cloud")
        }
}
```

## 2.6 *Suggestions History*

### 2.6.1 *Requisiti*

- **ViewModel e LiveData**

Vedi [2.2.1 Requisiti](#)

- **Glide**

Vedi [2.2.1 Requisiti](#)

- **Firebase Firestore e Storage**

Vedi [2.4.1 Requisiti](#)

- **RecyclerView**

Vedi [2.5.1 Requisiti](#)

- **Room**

La libreria **Room**, parte della categoria **Architettura in Android Jetpack**, rappresenta una soluzione dedicata alla persistenza dei dati su SQLite, caratterizzata da un approccio **ORM<sup>18</sup>**, cioè una tecnica di programmazione che fornisce, per l'appunto, mediante un'interfaccia orientata agli oggetti, tutti i servizi inerenti alla persistenza dei dati, astraendo nel contempo le caratteristiche implementative dello specifico RDBMS utilizzato.

Tecnicamente, si tratta di un **abstraction layer**, uno strato software che permette di sfruttare tutte le potenzialità del database senza la preoccupazione di affrontare ogni aspetto nei dettagli.

Le app che gestiscono quantità non banali di dati strutturati possono trarre grandi vantaggi dalla persistenza di tali dati localmente. Il caso d'uso più comune è memorizzare nella cache parti di dati rilevanti. In questo modo, quando il dispositivo non può accedere alla rete, l'utente può comunque visionare quel contenuto mentre è offline.

Il servizio Room può essere utilizzato da solo in un'applicazione o integrato con **ViewModel e LiveData** per supportare l'interfaccia utente nella massima efficienza.

Per garantire il corretto funzionamento di tale libreria, vi è necessità di implementare tre componenti specifici:

---

<sup>18</sup> Object-Relational Mapping

- **Room Database** → Rappresenta l’astrazione del database, identificabile dall’annotazione `@Database`. Contiene il supporto del database e funge da punto di accesso principale per la connessione ai dati relazionali persistenti dell’app.
- **Entity** → Ogni classe di questo tipo rappresenta una tabella del database. All’interno delle classi `@Entity` verranno predisposte tante variabili d’istanza quanti sono i campi previsti dallo schema della tabella.
- **DAO** → Un DAO<sup>19</sup> viene utilizzato per incamerare il codice che agirà sui dati, e conterrà i veri e propri comandi per le operazioni **CRUD**<sup>20</sup>, cioè operazioni che vengono eseguite durante la modellazione di un Database. Tipicamente esistono più DAO in un’applicazione, ma non è necessario ve ne siano tanti quante le Entità. In genere, ogni DAO raccoglie tutte le interazioni che riguardano un sottosistema del software: se stiamo modellando, ad esempio, l’accesso ai dati di una carrozzeria, potremo avere un DAO per gestire i dati utente, uno per il catalogo autoveicoli, uno per le parti di riparazione e così via.

## ● **Coroutines**

Una **Coroutine** è un modello di progettazione concorrenziale, introdotto in Kotlin alla versione 1.3, che è possibile utilizzare su Android per semplificare il codice che viene eseguito in modo asincrono.

Esse aiutano, inoltre, a gestire attività di lunga durata che potrebbero altrimenti bloccare il **thread**, cioè processo, principale e causare la mancata risposta dell’app. Senza dilungarsi troppo sui particolari, esse possono essere paragonate a dei “gestionali” che permettono l’avvio di particolari blocchi d’azioni su Thread separati, questi disponibili tramite dei **Dispatcher**, cioè collezioni di Thread, di cui ne esistono tre principali:

- **Main** → Lavoro sul Thread principale di UI, usato per azioni di bassa entità
- **IO** → Utilizzato per lavori Input Output di disco o di rete
- **Default** → Utilizzato per il lavoro intensivo della CPU

Essi possono lavorare sia in singolo o in coppia con un **Job**, cioè un’azione con un proprio ciclo di vita che inizia e termina alla fine di tale operazione.

Essi vengono definiti, specificatamente o automaticamente, all’interno di un **CoroutineScope**, cioè un Builder che si occupa di istanziare lo spazio dove devono essere avviati i lavori da intraprendere in uno specifico Thread.

Qui possono essere avviati blocchi generici di codice con la combinazione di **async** e **await**, oppure anche intere funzioni con l’utilizzo di **suspend**, cioè, ad esempio, se la funzione A viene avviata può essere appunto sospesa per permettere l’esecuzione di funzione B e, al termine, di essere ripresa per il completamento.

<sup>19</sup> Data Access Object

<sup>20</sup> Create, Read, Update, Delete

## 2.6.2 Funzionamento

Durante l'utilizzo dell'applicazione, l'utente potrebbe voler controllare quali cover, e di quale libro, siano già state suggerite su Anobii.

E' possibile così poter passare alla schermata inerente proprio alla cronologia delle cover suggerite in qualsiasi momento ed in qualsiasi situazione, cioè **Suggestions History**.

Tale schermata permette di visualizzare e gestire gli elementi al proprio interno similmente a come si è già visto nella schermata **Cloud Images**, infatti queste condividono lo stesso layout, con la sola differenza che quella attuale comprende un orientamento solo **Portrait** e non anche Landscape, per evitare una visualizzazione degli elementi poco leggibile.

Questa schermata e **Image Management** sono le uniche che condividono lo stesso **ViewModel**, per poter eseguire al meglio le operazioni di **Firebase** e il successivo suggerimento, salvando le informazioni accessibili da tale schermata.

Nel momento in cui l'utente deciderà di suggerire una cover, verrà prima verificato che sia presente una connessione ad internet per potersi collegare ai servizi online e successivamente sarà eseguita una ricerca all'interno di **Cloud Firestore** per sancire quale sia il numero dell'ultima richiesta inviata, così da non sovrascrivere i suggerimenti.

Dopodichè, verrà creata o aggiornata una collezione chiamata “**Books**” con all'interno il documento avente come nome il numero di richiesta effettuata e come campi tutte le informazioni inerenti alla cover suggerita. Per l'immagine stessa, invece, sarà utilizzato il nome della suddetta salvata all'interno del bucket principale in **Cloud Storage**, così da poterla reperire successivamente generando un Download URL utilizzabile da **Glide** per renderla visualizzabile all'interno dell'applicazione, esattamente come in Cloud Images.

A questo punto l'utente può decidere se visualizzare le informazioni del singolo elemento, eliminarlo o eliminare l'intera collezione di suggerimenti.

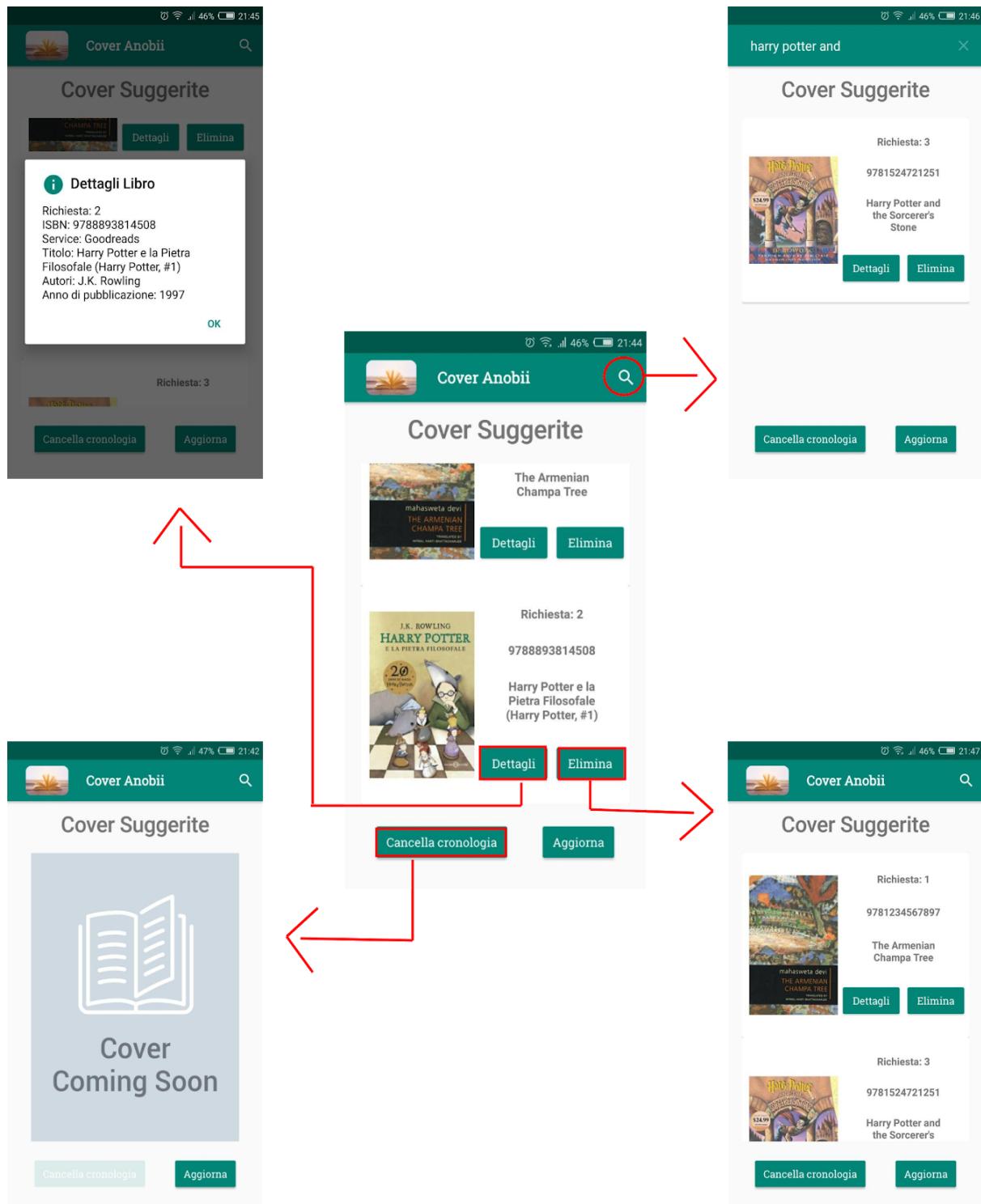
Possono esserci dei casi, però, in cui il collegamento con il server di Firestore non vada a buon fine, anche avendo una connessione ad internet attiva. In questo caso l'utente si troverebbe a non poter accedere alla cronologia e di non poter aggiungere alcun nuovo suggerimento, peggiorando l'esperienza di utilizzo complessiva.

Rende quindi necessario un utilizzo di salvataggio dati in locale per tenere conto dei suggerimenti che possono essere inviati in un secondo momento al server remoto.

Viene quindi utilizzata la libreria **Room** per poter salvare in un **Database locale** le stesse informazioni che sarebbero state inviate su Firebase, così da permettere all'utente di visualizzare la lista di elementi che vorrebbe fossero suggerite. Tali salvataggi verranno gestiti in modo asincrono su **Thread** differenti con l'utilizzo delle **Coroutines**, questo per permettere al programma di non bloccarsi durante le letture e le scritture al database.

Nel momento in cui il collegamento sarà ripristinato, l'utente dovrà reinserire le cover suggerite in precedenza nella sua esperienza in locale, questo per evitare un problema di ridondanza dei dati, in cui un utente remoto possa aver salvato una cover avente lo stesso numero di richiesta, sovrascrivendo lo stesso e creando errori di comprensione.

### 2.6.3 Esempi Layout



## 2.6.4 Esempi Codice

```
fun getListInfoBooks(application: Application) {
    _showProgressBar.value = true

    mFirestore.collection("collectionPath: Books").get().addOnSuccessListener { listDoc ->
        val data = arrayListOf<InfoBooksTable>()
        if (!listDoc.isEmpty()) {
            if (listDoc.documents.isNotEmpty()) {
                for (elem in listDoc.documents) {
                    data.add(InfoBooksTable(elem.id.toInt(), elem["isbn"].toString(),
                        elem["service"].toString(), elem["title"].toString(),
                        elem["author"].toString(), elem["publicationYear"].toString(),
                        elem["image"].toString()))
                }
            }
        }

        if (data.isEmpty()) _showPlaceholderNoImage.value = true else {
            _infoListImage.value = data
            _showPlaceholderNoImage.value = false
        }

        _showProgressBar.value = false
    }

    .addOnFailureListener { it: Exception
        uiScope.launch { thisCoroutineScope
            val data = InfoBooksDatabase.get(application).getInfoBooksDao.getBooks()
            as ArrayList<InfoBooksTable>

            if (data.isEmpty()) _showPlaceholderNoImage.value = true else {
                _infoListImage.value = data
                _showPlaceholderNoImage.value = false
            }
        }
    }
}
```

```
/**
 * In base alla ricerca effettuata, verranno identificati la lista di cover che combacia con
 * ciò che è stato richiesto, prima per id Richiesta e successivamente per tutti gli altri campi
 * Tali risultati verranno poi inseriti in una lista che aggiornerà la RecyclerView
 */
override fun onQueryTextChange(newText: String): Boolean {
    viewModel.infoListImage.observe(viewLifecycleOwner, Observer { it ArrayList<InfoBooksTable> })
    it?.let { listBook ->
        newList.let { input ->
            val inputSearch = input.toLowerCase(Locale.ROOT)

            val newList = arrayListOf<InfoBooksTable>()

            for (elem in listBook) {
                if (elem.idRequest.toString().contains(inputSearch)) {
                    newList.add(elem)
                } else if (elem.author.toLowerCase(Locale.ROOT).contains(inputSearch) || elem.title.toLowerCase(Locale.ROOT).contains(inputSearch) || elem.isbn.contains(inputSearch) || elem.service.toLowerCase(Locale.ROOT).contains(inputSearch) || elem.publicationYear.contains(inputSearch)) {
                    newList.add(elem)
                }
            }
            adapter.updateList(newList)
        }
    }
}
```

```
@Database(version = 1, entities = [InfoBooksTable::class], exportSchema = false)
abstract class InfoBooksDatabase : RoomDatabase() {

    // Dao di Room Database
    abstract val getInfoBooksDao: InfoBooksDao

    // Tutto ciò al suo interno viene istanziato una volta sola
    companion object {

        // L'istanza non verrà salvata in cache
        @Volatile
        private var INSTANCE: InfoBooksDatabase? = null

        fun get(application: Application): InfoBooksDatabase {
            // L'interno non verrà mai eseguito contemporaneamente su due thread diversi
            synchronized(lock) {
                var instance = INSTANCE

                // Non permette l'istanza multiple dello stesso database
                if (instance == null) {
                    instance = Room
                        .databaseBuilder(application, InfoBooksDatabase::class.java,
                            name = "infoBooks")
                        .fallbackToDestructiveMigration()
                        .build()

                    INSTANCE = instance
                }
            }

            return instance
        }
    }
}
```

```
/**
 * Info Books Dao
 * Interfaccia DAO, cioè Data Access Object, in grado di comunicare al database Room utilizzando
 * il Linguaggio SQL
 * Author Simone Tugnetti - Ovalab
 */
@Dao
interface InfoBooksDao {

    // Funzione suspend, usata per comunicare su thread differenti tramite le Coroutines
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertBook(book: InfoBooksTable)

    @Query(value = "select * from InfoBooksTable")
    suspend fun getBooks(): List<InfoBooksTable>

    @Query(value = "select * from InfoBooksTable where idRequest = :request")
    suspend fun getSpecificBook(request: Int): InfoBooksTable?

    @Query(value = "delete from InfoBooksTable")
    suspend fun deleteAllBooks()

    @Query(value = "delete from InfoBooksTable where idRequest = :request")
    suspend fun deleteBook(request: Int)
}
```

## 2.7 *Linguaggi e Tool Utilizzati*

Precedentemente si è descritto questo programma come un'applicazione **Android** nativa sviluppata in **Kotlin**. Android, però, è il sistema operativo al quale l'app è collegata per permetterne il corretto funzionamento, vi è quindi bisogno di un ambiente di sviluppo consono a rendere i requisiti ed il funzionamento di ogni parte dell'app sopra descritta tangibili.

Viene quindi utilizzato **Android Studio**. Nello specifico, si tratta di un **IDE**<sup>21</sup>, cioè un software che, in fase di programmazione, supporta i programmatore nello sviluppo del codice sorgente di un programma, pubblicato nel dicembre del 2014 e basato sul software di **JetBrains IntelliJ IDEA**.

Esso è stato progettato specificamente per lo sviluppo di applicazioni Android, sostituendo gli **ADT**<sup>22</sup> del principale predecessore riguardo tale sviluppo, cioè **Eclipse**, diventando l'IDE primario di Google per lo sviluppo nativo di tali applicazioni.

Oltre ad essere un potente editor e a possedere strumenti di sviluppo di IntelliJ, Android Studio offre ancora più funzionalità che migliorano la produttività durante la creazione delle app, ad esempio un sistema di generazione flessibile basato su **Gradle**, un emulatore veloce e ricco di funzionalità, integrazione con **GitHub** per condividere il lavoro con un team, ecc.... Il principale linguaggio di programmazione concepito inizialmente per questo IDE era **Java**, un linguaggio ad alto livello, orientato agli oggetti e a tipizzazione statica, che si appoggia sull'omonima piattaforma software di esecuzione, specificamente progettato per essere il più possibile indipendente dalla piattaforma hardware di esecuzione, infatti il prodotto della compilazione è in un formato chiamato **bytecode** che può essere eseguito da una qualunque implementazione di un processore virtuale detto **JVM**<sup>23</sup>.

Vi è, però, un ulteriore linguaggio di programmazione, protagonista dello sviluppo della suddetta app, cioè **Kotlin**, integrato nell'ambiente di sviluppo Android Studio a partire dalla versione 3.0.

Kotlin è un linguaggio di programmazione general purpose ed open source sviluppato dalla stessa JetBrains. Esso si basa sulla JVM ed è ispirato ad altri linguaggi di programmazione tra i quali Scala e lo stesso Java, mentre ulteriori spunti sintattici sono stati presi da linguaggi classici, come il Pascal e moderni come Go o F#.

Kotlin è strutturato per interoperare con la piattaforma **JRE**<sup>24</sup> come target principale, il che garantisce il funzionamento delle applicazioni in ogni ambiente che accetti la JVM, ivi compreso Android.

Dal 7 maggio 2019, esso è infatti il linguaggio consigliato da Google per lo sviluppo di tali applicazioni.

---

<sup>21</sup> Integrated Development Environment

<sup>22</sup> Android Development Tools

<sup>23</sup> Java Virtual Machine

<sup>24</sup> Java Runtime Environment

### 3. Conclusioni

Si è quindi arrivati alla fine del progetto, dopo non poche difficoltà, tempo, blocchi e studio impiegato per rendere reale tutto quello elencato finora.

Durante questo periodo svolto in stage, sono riuscito ad archiviare le informazioni base generiche riguardo questo nuovo linguaggio di programmazione, quale Kotlin, e al suo relativo utilizzo all'interno dell'ambiente di sviluppo Android e non solo.

Oltre, infatti, l'utilizzo su questo sistema operativo, ho anche imparato qualche tecnica base di programmazione, cioè l'utilizzo delle Scope Functions, come vengono create e gestite chiamate a servizi remoti con l'utilizzo del sistema REST, l'utilizzo di funzioni, classi, gerarchie, ecc....

Nonostante avessi avuto una formazione inherente al linguaggio Java sviluppando programmi nativi ed applicazioni Android, il passaggio al linguaggio Kotlin non è stato così difficoltoso come me l'immaginavo. Tutt'altro, la comprensione complessiva è sembrata molto leggera e reattiva, se non forse per alcuni argomenti legati alla gestioni di azioni asincrone, data anche per la conoscenza di alcuni linguaggi avente una sintassi molto simile, come C e Swift, ed una metodologia orientata agli oggetti, come per il sopra citato Java e Python.

Doveva però essere trovato un modo per rendere condivisibile il lavoro svolto. Qui entra in gioco GitLab, un servizio che permette, per l'appunto, la condivisione del lavoro svolto con altri membri del team e, nonostante l'avessi già utilizzato in precedenza, rimane sempre complicato capirne il funzionamento, con commit eseguiti male, push dimenticati e commenti non inseriti, ma alla fine, tralasciando piccoli particolari, le informazioni base sono state recepite senza problemi e la comprensione è risultata piuttosto leggera.

Questo progetto è risultato essere il più grande ed ambizioso che abbia mai creato, nonché uno dei primissimi realizzati in Kotlin, avente il collegamento con servizi online, sistemi di fallback, stili e liste dinamiche.

La realizzazione di tutto ciò ha avuto perlopiù momenti duri e difficoltosi dati dallo studio e dall'implementazione di argomenti che finora mi erano del tutto nuovi, come quando l'intero programma può smettere completamente di funzionare per colpa di un errore alla riga 135 colonna 7, ma questi per nulla impossibili e questo progetto finale ne è la prova.

La prova di essere riuscito ad implementare e collegare con un filo logico tutti questi argomenti insieme per creare qualcosa di ottimale e funzionale, in concomitanza con la mia passione personale per l'ambito mobile, rendendomi orgoglioso del risultato finale.

Spero che anche voi che leggete questo risultato riusciate a scorgere la fatica e l'impegno nell'essere riuscito a creare qualcosa di così ambizioso, così da provare la stessa soddisfazione che ho provato anche io.

## 4. Sitologia

JC MAXWELL

<https://www.jcmaxwell.edu.it/>

ITS-ICT Piemonte

<https://www.its-ictpiemonte.it/>

Ovolab S.r.l

<https://www.ovolab.com/>

Android Jetpack

<https://developers-it.googleblog.com/2018/05/android-jetpack-per-accelerare-lo.html>

<https://developer.android.com/topic/libraries/support-library>

<https://developer.android.com/topic/libraries/architecture>

<https://developer.android.com/kotlin/ktx>

Navigation

<https://developer.android.com/guide/navigation>

ViewModel

<https://developer.android.com/topic/libraries/architecture/viewmodel>

<https://developer.android.com/reference/androidx/lifecycle/LiveData>

Data Binding

<https://developer.android.com/topic/libraries/data-binding>

Retrofit

<https://square.github.io/retrofit/>

<https://www.html.it/pag/307781/la-libreria-retrofit/>

[https://it.wikipedia.org/wiki/Representational\\_State\\_Transfer](https://it.wikipedia.org/wiki/Representational_State_Transfer)

Glide

<https://bumptech.github.io/glide/>

Dexter

<https://github.com/Karumi/Dexter>

Permessi

<https://www.html.it/pag/370592/gestione-dei-permessi-su-android/>

## API

<https://www.redhat.com/it/topics/api/what-are-application-programming-interfaces>

## Google Play Services Vision

<https://developers.google.com/vision/introduction>

<https://developer.android.com/reference/android/view/SurfaceView>

## Firebase

<https://firebase.google.com/>

<https://www.html.it/articoli/firebase/>

<https://it.wikipedia.org/wiki/NoSQL>

## Cloud Firestore

<https://firebase.google.com/docs/firestore>

## Cloud Storage

<https://firebase.google.com/docs/storage>

## RecyclerView

<https://www.html.it/articoli/recyclerview-dietro-le-quinte/>

## Room

<https://developer.android.com/training/data-storage/room>

<https://www.html.it/pag/71033/room-orm-su-sqlite/>

## Coroutines

<https://developer.android.com/kotlin/coroutines>

<https://medium.com/androiddevelopers/coroutines-on-android-part-i-getting-the-background-3e0e54d20bb>

<https://medium.com/@elye.project/understanding-suspend-function-of-coroutines-de26b070c5ed>

## Android Studio

<https://developer.android.com/studio/intro>

[https://it.wikipedia.org/wiki/Android\\_Studio](https://it.wikipedia.org/wiki/Android_Studio)

## Java

[https://it.wikipedia.org/wiki/Java\\_\(linguaggio\\_di\\_programmazione\)](https://it.wikipedia.org/wiki/Java_(linguaggio_di_programmazione))

## Kotlin

<https://kotlinlang.org/>

[https://it.wikipedia.org/wiki/Kotlin\\_\(linguaggio\\_di\\_programmazione\)](https://it.wikipedia.org/wiki/Kotlin_(linguaggio_di_programmazione))

## 5. Ringraziamenti

Ringrazio il relatore **Marco Roberti** per avermi seguito durante la stesura di questa tesina.

Ringrazio l'istituto **ITS-ICT Piemonte** per l'opportunità di seguire il percorso formativo Web & Mobile App Development.

Ringrazio l'azienda ospitante **Ovolab S.r.l** per avermi concesso l'opportunità di sostenere il periodo di stage curriculare.

Infine, ringrazio tutti gli **amici e parenti** che mi sono stati vicino durante questi due anni di formazione e con il relativo stage.